MAS6024 Assignment 3 - 190186373

Part 1

To simulate making a move in the game, we can create a function which takes the starting position as an input and return the new position as the output. We also allow the user to specify λ. λ takes the default value 1.

We can sample the vector `direction` and use if-statements to simulate a game move in one direction. To account for rule 4 about X being large than the available spaces in a specified direction, an additional if-statement is included. Note that `to_print` has also been included as a Boolean function argument to keep track of how the function works. The code and pseudo-code for this function `make_move` is described below, taking arguments `x1` and `y1` as the starting position on `lgrid`.

Pseudo-code for `make_move`

```
make_move <- function(x1, y1, lambda = 1, to_print = FALSE)
  X is a poisson random variable, with parameter lambda
  direction is 'left, right, up, or down' all equally likely.

  if direction is left
    if X too large, force X to take value that moves y1 to 1
    move X spaces left
    record new location

  if direction is right
    if X too large, force X to take value that moves y1 to 8
    move X spaces left
    record new location

  if direction is up
    if X too large, force X to take value that moves x1 to 1
    move X spaces left
    record new location

  if direction is up
    if X too large, force X to take value that moves x1 to 8
    move X spaces left
    record new location
  return(new location)
```

```r
make_move <- function(loc, lambda = 1, to_print = FALSE){
  X <- rpois(1, lambda)
  direction <- sample(c('up', 'down', 'left', 'right'), 1)

  if(direction == 'left'){
    if(loc[2] - X < 1) X <- loc[2] - 1
    if(to_print == TRUE) cat('Moving left', X, 'squares \n')
    new_loc <- c(loc[1], loc[2] - X)}

  if(direction == 'right'){
    if(loc[2] + X > 8) X <- 8 - loc[2]
    if(to_print == TRUE) cat('Moving right', X, 'squares \n')
    new_loc <- c(loc[1], loc[2] + X)}

  if(direction == 'up'){
    if(loc[1] - X < 1) X <- loc[1] - 1
    if(to_print == TRUE) cat('Moving up', X, 'squares \n')
    new_loc <- c(loc[1] - X, loc[2])}

  if(direction == 'down'){
    if(loc[1] + X > 8) X <- 8 - loc[1]
    if(to_print == TRUE) cat('Moving down', X, 'squares \n')
    new_loc <- c(loc[1] + X, loc[2])}
  return(new_loc)
  }
```

Here is an example of a move, starting at from $(4, 4)$.

```r
make_move(loc = c(4,4), lambda = 3, to_print = T)
```

```
## Moving right 4 squares

## [1] 4 8
```

Part 2

Now we can create the whole game. The game is designed such that it relies on a while loop to iteratively make moves and pick up letters. Let use define some objects that will prove useful in creating the game:

- `start_loc` - random starting position,
- `next_loc` - the position after a move is completed,
- `num_moves` - number of moves, including the initial move onto the grid,
- `letters` - vector of letters that we have picked up,
- `cat` - vector of the letters c, a and t.

Psuedo-code for game

```
game <- function(lambda = 1, to_print = FALSE)
  completed 0 moves so far
  randomly choose x position
  randomly choose y position
  add first letter to letter vector
  game has finished = FALSE

  while game not finished
    add 1 to number of moves made
    call make_move to get new position
    record letter of new position
    if this letter isn't the letter vector, add it
    if c, a and t are in the letter vector, game has finished = TRUE
    else, let the current position be the new starting position and complete another iteration
  return(total number of moves)
```

```
game <- function(lambda = 1, to_print = FALSE){
  num_moves <- 0
  start_loc <- c(sample(dim(lgrid)[1], 1), sample(dim(lgrid)[2], 1))
  letters <- c(lgrid[start_loc[1], start_loc[2]])
  cat <- c("c", "a", "t")
  if(to_print == TRUE) cat('Start', start_loc, '\n', letters, '\n')
  finished_game = FALSE

  while(finished_game == FALSE){
    num_moves <- num_moves + 1
    next_loc <- make_move(start_loc)
    letter <- lgrid[next_loc[1], next_loc[2]]
    if(any(letters == letter) == FALSE) letters <- c(letters, letter)
    if(to_print == T) cat('Move #', num_moves, 'Moving from', start_loc,
                          'to', next_loc, '\n', letters, '\n')
    if(all(cat %in% letters)) finished_game = TRUE else{
      start_loc <- next_loc}
  }
  if(to_print == T) cat('\n Game complete!')
  return(num_moves)
}
```

We test if a letter is in the letter vector using `any(letters == letter)`. The game finishes when "c", "a", "t" is in `letters`. In the code, this translates to `if(all(cat %in% letters)) finished_game = TRUE`, where `%in%` outputs true or false, depending on whether the letter is already in `letters` or not. The `all` function outputs true or false, depending on if all elements of `cat` are elements of `letters`.

Here is an example of one run of the game, with the first and last three moves to demonstrate its functionality.

```
Start 5 4
 f
Move # 1 Moving from 5 4 to 5 4
 f
Move # 2 Moving from 5 4 to 5 5
 f k

Move # 110 Moving from 2 5 to 2 6
 f k h p d s q e o u x r l y i a z c g w
Move # 111 Moving from 2 6 to 2 6
 f k h p d s q e o u x r l y i a z c g w
Move # 112 Moving from 2 6 to 2 7
 f k h p d s q e o u x r l y i a z c g w t

 Game complete![1] 112
```

Part 3

- Relate the number of replicates to the accuracy of the estimators and possibly link it to a specific number of decimal points of agreement.

Since the function for our game returns the number of moves required to complete an iteration of the game, we can replicate the game a suitable amount of times for $\lambda = 1$ and $\lambda = 4$. It is important to maximise the number of replications to improve the accuracy of the numerical summaries of the distribution. 30,000 replications for one case takes about one minute.

```r
m <- 30000
rep_lambda1 <- replicate(n = m, expr = game(lambda = 1))
rep_lambda4 <- replicate(n = m, expr = game(lambda = 4))
```

Let us evaluate the stability of the distribution by comparing it with another simulation of the same function. Hopefully this will justify our choice of number of replications.

```r
compare_dist <- replicate(n = m, expr = game(lambda = 1))
compare_table <- table(compare_dist)/m # probabilities of each unique number of moves
lambda1_table <- table(rep_lambda1)/m
num_moves_in_common <- min(length(compare_table), length(lambda1_table))
mean_diff_prob <- mean(abs(compare_table[1:num_moves_in_common] - lambda1_table[1:num_moves_in_common])
max_prob_diff <- max(abs(compare_table[1:num_moves_in_common] - lambda1_table[1:num_moves_in_common]))
length(rep_lambda1[rep_lambda1 > 600])/m
```

```
## [1] 0.005933333
```

```r
cat('Comparing two lambda = 1 distributions \n| Difference in minimum values:', min(rep_lambda1) - min(
    '\n| Difference in maximum value:', abs(max(rep_lambda1)-max(compare_dist)),
    '\n| Difference in means:', abs(mean(rep_lambda1)- mean(compare_dist)),
    '\n| Difference in sd', abs(sd(rep_lambda1)-sd(compare_dist)),
    '\n| Mean difference in probabilities for each game length:', mean_diff_prob,
    '\n| Max difference in probability for a game length:', max_prob_diff)
```

```
## Comparing two lambda = 1 distributions
## | Difference in minimum values: 0
## | Difference in maximum value: 288
## | Difference in means: 1.228867
## | Difference in sd 3.07675
## | Mean difference in probabilities for each game length: 0.0001893091
## | Max difference in probability for a game length: 0.0012
```

The difference in the mean number of moves is only 1 which seems acceptable. Ideally, we would want the means to agree to at least 1 decimal place but it would likely require over 100,000 replications. Time does not permit this. The right tail of the distribution seems more unstable with 65 difference in the maximum number of moves. Despite this, less than 1% of games take more than 600 moves so we can accept this difference. Furthermore, the mean difference in probability for a unique game length is under 0.1%, so 30,000 replications is acceptable.

```r
cat('Game with lambda = 1 \n | Minimum:', min(rep_lambda1), '| Maximum:',
    max(rep_lambda1), '| Mean:', mean(rep_lambda1), '| Stan. Dev.', sd(rep_lambda1), '\n')
```

```
## Game with lambda = 1
##  | Minimum: 2 | Maximum: 1528 | Mean: 137.477 | Stan. Dev. 122.0835
```

```r
quantile(rep_lambda1, c(0.025, 0.25, 0.5, 0.75, 0.975))
```

```
##  2.5%   25%   50%   75% 97.5%
##     7    47   104   192   459
```

```r
cat('Game with lambda = 4 \n | Minimum:', min(rep_lambda4), '| Maximum:',
    max(rep_lambda4), '| Mean:', mean(rep_lambda4), '| Stan. Dev.', sd(rep_lambda4), '\n')
```

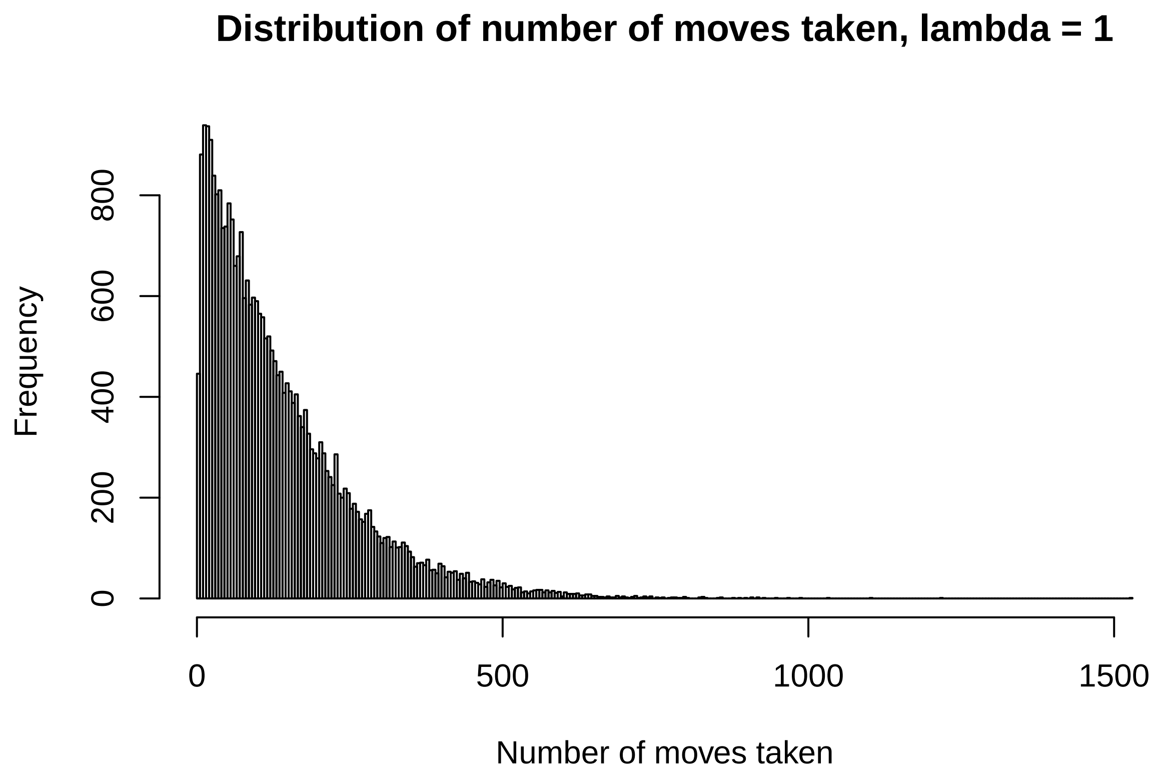
```
## Game with lambda = 4
##  | Minimum: 2 | Maximum: 1297 | Mean: 137.8331 | Stan. Dev. 122.2065
```

```r
quantile(rep_lambda4, c(0.025, 0.25, 0.5, 0.75, 0.975))
```

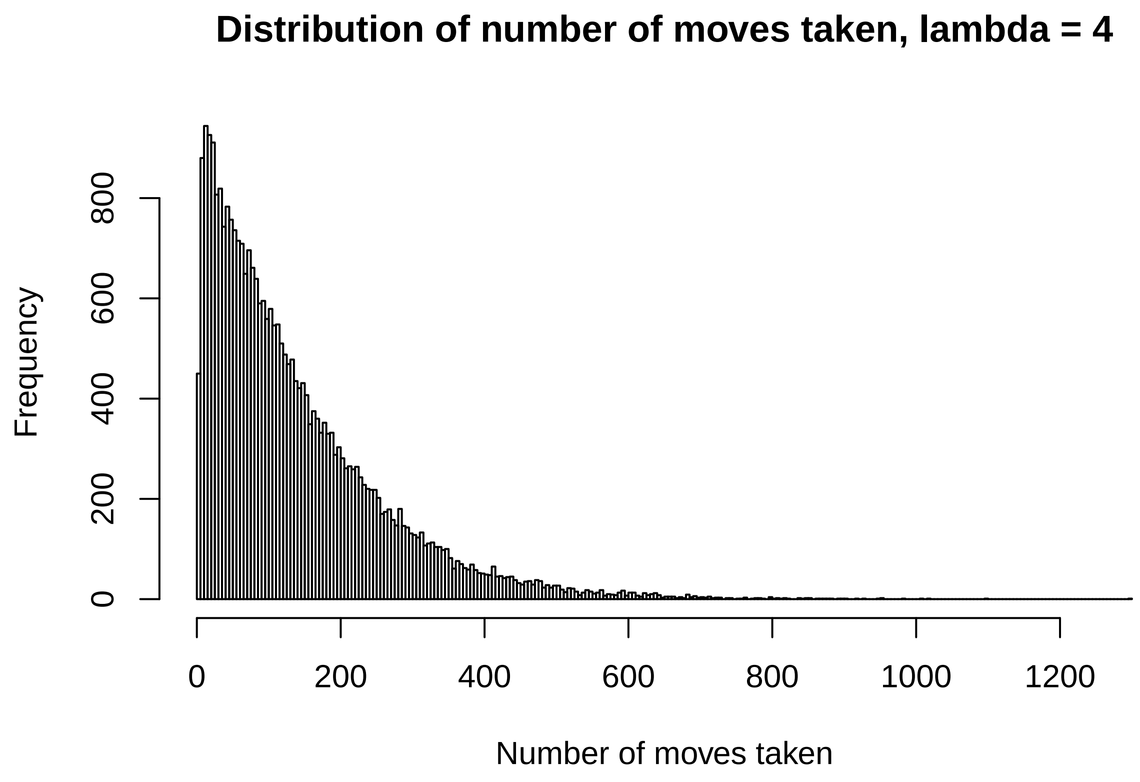
```
##  2.5%   25%   50%   75% 97.5%
##     7    47   104   194   458
```

At first glance, the probability distributions for $\lambda = 1$ and $\lambda = 4$ seem similar, with 50% of our games lasting under 2 moves. Both distributions have large variation.

```r
hist(rep_lambda1, m/100,
     main = "Distribution of number of moves taken, lambda = 1",
     xlab = "Number of moves taken")
```



```r
hist(rep_lambda4, m/100,
     main = "Distribution of number of moves taken, lambda = 4",
     xlab = "Number of moves taken")
```



Part 4

The most efficient way to modify the game to suit the new rules is to create a function that implements the new rule and call this function in the main code. Using a for loop, we can iteratively change each letter to its succeeding letter using a vector containing each letter of the alphabet, `alphabet`. This is done by subsetting `alphabet`. The psuedo-code and code for `letter_rotate` is shown below. Note that an if statement to force any ‘z’ to ‘a’ is also included.

```
letter_rotate <- function(letters vector)
  create alphabet vector
  for i in 1:length(letters vector)
    if letters[i] is z, change it to a
    else, force each letter to be the next letter in the alphabet
  return new letters vector
```

```r
letter_rotate <- function(letters){
  alphabet <- c('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
                'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z')
  for(i in 1:length(letters)){
    if(letters[i] == 'z') letters[i] <- 'a'
    else letters[i] <- alphabet[i + 1]
  }
  return(letters)
}
```

We can now input our function `letter_rotate` into our game. The game modifications are separated by multiple '#' for clarity. A further if statement is created to implement the consequences of landing on a green square. The co-ordinates of the two green squares according to `lgrid` indexing are $(2, 6)$ and $(3, 7)$. The if statement `if((next_loc == c(2, 6)) || (next_loc == c(3, 7)))` ensures that if the new x and y values are either of the two green squares, we call `letter_rotate`. The new game `game_mod` is shown below.

```
if new square is (2, 6) or (3, 7)
  change each letter in vector to succeeding letter.
```

```
game_mod <- function(lambda = 1, to_print = FALSE){
  num_moves <- 0
  start_loc <- c(sample(dim(lgrid)[1], 1), sample(dim(lgrid)[2], 1))
  letters <- c(lgrid[start_loc[1], start_loc[2]])
  cat <- c("c", "a", "t")
  if(to_print == TRUE) cat('Start', start_loc, '\n', letters, '\n')
  finished_game = FALSE

  while(finished_game == FALSE){
    num_moves <- num_moves + 1
    next_loc <- make_move(start_loc)
    letter <- lgrid[next_loc[1], next_loc[2]]
    if(any(letters == letter) == FALSE) letters <- c(letters, letter)
    ############# NEW RULE ###########
    if((next_loc == c(2, 6)) || (next_loc == c(3, 7))){
      if(to_print == T) cat('\n Moving onto green square! \n')
      letters <- letter_rotate(letters)}
    ##################################
    if(to_print == T) cat('Move #', num_moves, 'Moving from', start_loc,
                          'to', next_loc, '\n', letters, '\n')
    if(all(cat %in% letters)) finished_game = TRUE else{
      start_loc <- next_loc
    }
  }
  if(to_print == T) cat('\n Game complete!')
  return(num_moves)
}
```

An example of how the new rules change the game is demonstrated below with some of the printed output from a run of the game.

```
Move # 134 Moving from 2 8 to 2 8
 b d w n j l v c a r m s e z y f p i

 Moving onto green square!
Move # 135 Moving from 2 8 to 2 6
 c e x o k m w d b s n t f a z g q j

 Game complete!
```

Part 5

```r
#rep_lambda1
rep_mod <- replicate(n = m, expr = game_mod(lambda = 1))
```

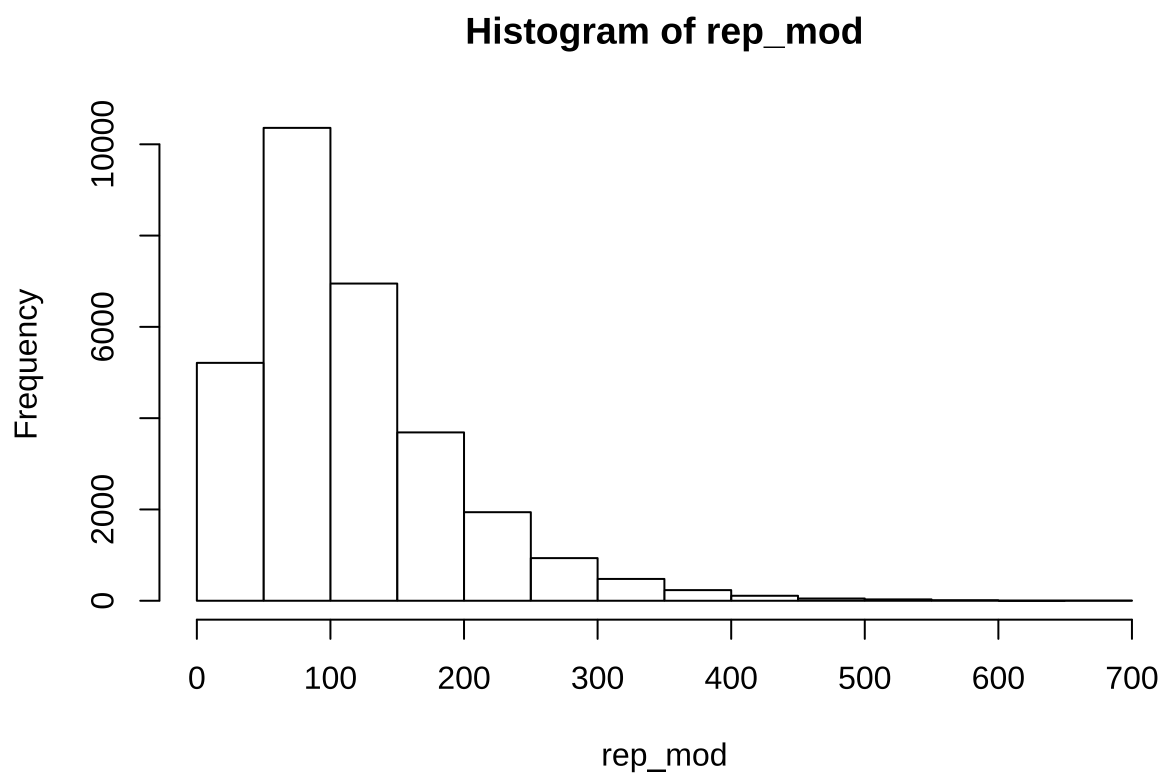
```r
cat('Modified game with lambda = 1 \n | Minimum:', min(rep_mod), '| Maximum:',
    max(rep_mod), '| Mean:', mean(rep_mod), '| Stan. Dev.', sd(rep_mod), '\n')
```

```
## Modified game with lambda = 1
##  | Minimum: 2 | Maximum: 697 | Mean: 115.1314 | Stan. Dev. 77.30369
```

```r
quantile(rep_mod, c(0.025, 0.25, 0.5, 0.75, 0.975))
```

```
##  2.5%   25%   50%   75% 97.5%
##    14    61    97   150   314
```

```r
hist(rep_mod)
```



```r
#plot(rep_mod, col="red", type = "h", main = "", xlab = "", ylab = "")
#plot(rep_lambda1,col="green", type = "")
```